AD-A079 276 MARYLAND UNIV COLLEGE PARK COMPUTER SCIENCE CENTER F/G 9/2
FUNCTIONAL SPECIFICATION OF ASYNCHRONOUS PROCESSES AND ITS APPL--ETC(U)
MAR 79 P ZAVE N00014-77-C-0623
UNCLASSIFIED CSC-TR-775 NL

END
DATE FILMED
2-80
DDC

UNIVERSITY OF MARYLAND
COMPUTER SCIENCE CENTER

COLLEGE PARK, MARYLAND
20742

(1)

(14) CSC-TR-775

(11) March 1979

(6) FUNCTIONAL SPECIFICATION OF ASYNCHRONOUS PROCESSES
AND ITS APPLICATION TO
THE EARLY PHASES OF SYSTEM DEVELOPMENT*,

401 052

(10) Pamela Zave
Department of Computer Science
University of Maryland
College Park, Maryland 20742

(12) 37

(15) N00014-77-C-0623

(9) Technical Report

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

*This research was supported in part by the Office of Naval Research, Mathematics Branch, under Contract N00014-77-C-0623, and by the Computer Science Center, University of Maryland, College Park.

403 018

slt

0. Abstract

This paper introduces a functional language for system specification, and shows how it can be extended to the domain of asynchronously interacting processes. The language has many desirable properties for design specification, and is also an effective vehicle for the specification of requirements. It is argued that the primitive concepts of this language are basic building blocks that can support a methodology in which all system development phases use the same language and are related to one another by well-structured elaborations.

Index Terms: system specification

parallel processes

requirements

high-level design

C.R. Categories: 4.00, 6.00, 8.10

FUNCTIONAL SPECIFICATION OF ASYNCHRONOUS PROCESSES AND ITS APPLICATION TO THE EARLY PHASES OF SYSTEM DEVELOPMENT

1. Introduction

Serious problems often arise during the development of a large, complex digital system. Problems caused by poor requirements analysis or faulty early design decisions are among the worst because they are often propagated throughout the design.

Requirements analysis often yields a poor description of the required system. Specifications of requirements can be informal, ambiguous, incomplete, inconsistent, and difficult to understand. This leads to situations in which either (a) inconsistencies and ambiguities are not found until the detailed-design or implementation phases, necessitating costly revisions, or (b) the final product does not meet the customer's needs. This could happen because of poor communication between the designers and the customer, or because the customer did not understand his own needs (and the designers could not help him).

Another major source of trouble is the translation from requirements to an early (high-level) design, because this is a transition between incommensurate descriptions of the system. Stimulus-response paths from the requirements must be mapped onto processor and storage structures, taking into account problems of performance, resource contention, synchronization, reliability, etc. It is often difficult to find even one solution to satisfy all the constraints, let alone a good one--a solution in which well-chosen structures generate robust yet flexible behavior, and cost-effective performance trades have been made.

This paper introduces some new ideas on how specifications can alleviate these problems, as well as a specification language to formalize the intuitive notions. The two major features of this language are that (a) it is functional, and (b) its basic entities are asynchronously interacting processes.

Section 2 presents functional notation as a familiar, yet precise, vehicle for communication about systems. It permits much automated

consistency checking, can specify any level of abstraction (or detail), and can be simulated. Section 3 shows how purely functional specification techniques can be extended to the domain of asynchronously interacting processes, making functional specification of whole systems possible.

An example of system specification in Section 4 shows that our extended functional notation can be used to describe high-level designs in a natural and elegant way. Even more important, it can be used to specify system requirements as well. This is done by specifying the required system as a set of digital processes which interact with another set of processes that are digital simulations of the system's environment.

This form of requirements specification offers many advantages. Some of them accrue from the application of good design tools and techniques earlier in the development process. The requirements themselves, for instance, can now be simulated. Thus they can be tested by the designers, and can also generate behaviors to be criticized by the customer. This allows the customer to participate in early system development in a practical and constructive way.

Another advantage is that by thinking in terms of interacting entities (system and environment processes) from the beginning, analysts and designers are encouraged to consider performance and resource requirements as integral parts of the development. These requirements are crucial, but easily neglected in the rush to clarify the "logical functions" of the system. Yet in a real-time system, a late response to a stimulus may be as bad as one which is "logically" wrong.

Finally, this approach allows all phases of development to use the same notation. Furthermore, the number of primitive concepts needed is small enough to be defended as a set of "basic building blocks". This makes it conceivable that system development could someday become a sequence of modest elaborations--based on thorough understanding of these building blocks--rather than great leaps of the imagination.

2. Functional Specifications

Figure 1 shows an initial functional specification of a stack as a collection of sets and functions. When implemented, a primitive set becomes a data structure capable of representing elements in that set, and a primitive function becomes a procedure.

Sets can be primitives, as "STACK" and "ELEMENT" are in Figure 1, or can be defined in terms of "smaller" primitives, using the syntax:

```
SET-NAME =
     "set-expression".
```

In Figure 1, the set "ACKNOWLEDGMENT" is defined by enumerating its members, and "ANSWER" is defined as a union of sets.

A function is always declared by naming its domain and range sets, as in:

```
function-name:
     DOMAIN-SET-ONE x DOMAIN-SET-TWO
---> RANGE-SET.
```

If the function is not a primitive, it is then defined by a functional expression using arguments, constants, and other functions. For instance, the expression defining "push" is:

```
push(s,e) =
     [full(s):  (s,refused),
      true    :  (put-on(s,e),accepted)
     ].
```

This expression (based on a LISP-like constructor for conditional evaluation) says that if the stack (argument s) is full, then the value of "push" is the same stack with the acknowledgment "refused". Otherwise it is the result of "put-on(s,e)" (argument e is the new element) and the acknowledgment "accepted".

Several advantages of this method of specification for requirements and high-level designs follow immediately:

(a) It is formal, and can be applied at any level of abstraction, depending only on the primitives used--making it attractive as a vehicle of communication among all those participating in a development effort.

(b) Functional notation describes an input/output relation without binding the data or control structures used to compute it. This minimizes the danger of confusing design and implementation decisions.

(c) The syntax is simple, yet incorporates strong typing, making syntax checking a powerful means of error detection.

Figure 1. Initial specification of a stack.
Set-defining expressions use union (U), cross product (x), and enumeration of constants ({a,b}). Function-defining expressions use tuple formation ((f,g)), composition (f(g(h))), and selection ([p1:f1, p2:f2, ... pn:fn], evaluated as the first function fi such that the predicate pi is true).
Stylistic conventions are that set names are in capitals, function names are in lower case, and non-numeric constants are underlined.

```
ACKNOWLEDGMENT =
     { accepted, refused }

ANSWER =
     ELEMENT U { stack-is-empty }

push:
     STACK x ELEMENT
---> STACK x ACKNOWLEDGMENT

push(s,e) =
     [full(s):  (s,refused),
      true    :  (put-on(s,e),accepted)
     ]

full:
     STACK
---> { true, false }

put-on:
     STACK x ELEMENT
---> STACK

pop:
     STACK
---> STACK x ANSWER

pop(s) =
     [empty(s):  (s,stack-is-empty),
      true     :  take-off(s)
     ]

empty:
     STACK
---> { true, false }

take-off:
     STACK
---> STACK x ELEMENT
```

(d) Functional notation precludes the use of side effects, so that the designer is forced to consider explicitly all possible domain values and their consequences. Thus as soon as "push" is declared as a primitive function we know that it produces, as a component of its value, an "acknowledgment", expressing the fact that a "push" may fail because the stack is full.

Attention to this kind of detail is essential in the management of a large development effort, in which each person involved can only be familiar with the modules for which he is responsible. In such situations, nothing is as important as precise and complete interface specification.

In the example, failure of a "push" or "pop" always produces a message to that effect. This message will be sent back to the party requesting the operation (see Section 4), which can then take appropriate action. This places responsibility for error recovery on the party that presumably understands the significance of the problem ([Goodenough 75]). This form of error analysis and recovery is a strong point of HOS ([Hamilton & Zeldin 76]).

(e) Finally, functional notation is "effective" (or executable). Even though functional operations such as composition do not specify exactly how one function is to be evaluated and its value passed to the next, their structures are easily simulated by an interpreter. This alleviates the problem that requirements must be approved long before there is an implementation to generate the behavior implied by them: requirements specifications in functional form can be simulated during the requirements phase. This is particularly valuable in providing early feedback to the customer.

For a specification to be interpreted, some implementation of each primitive must be supplied to the interpreter. The logical properties of a system, for instance, could be tested by a simulation in which most primitive functions evaluate to arbitrary values in their ranges. By giving predicates both "true" and "false" values, etc., the designers can exercise all control paths and check on their interactions. Primitive functions which are evaluated by asking for a decision from a terminal automatically constitute an interactive testing system for use by designers and customers. Performance properties, on the other hand, can be simulated by using primitive functions which mainly record their estimated resource consumptions. Summary data can then provide accurate global estimates. Both logical and performance simulation have been used successfully in the SREM project ([Alford 77], [Bell et al. 77]).

Specification techniques for data abstractions ([Liskov & Zilles 75]) likewise describe data structures such as stacks, and all the legal operations on them, in a formal and communicable way. But there are important differences between the goals, and hence the directions, of work on these two

types of specifications.

Figure 1 is not a data abstraction specification because it does nothing to capture the concept of "stackness", one property of which is expressed by the axiom:

stack(s) & integer(i) --> pop(push(s,i)) = s.

The idea behind such specifications is to use axioms, graphs, or some other such non-procedural medium to capture all the properties that pertain to the concept of the data abstraction in the designer's mind, and then to prove that a programmed implementation is equivalent to the formal specification. The crucial property of data abstraction specifications is what Liskov and Zilles call "minimality": because it does not bias or constrain the implementation of an abstract stack in any way, the ideal stack specification describes equally well any valid implementation of "stackness".

Our specification language must be different because, in the early phases of development, the eventual implementation of an abstract concept is not a matter of indifference. It will be the result of difficult design choices, usually concerning cost/performance trades; the specification language must record, rather than avoid, these decisions.

Functional notation records decisions well because there are natural units to which performance and resource requirements can be attached, and natural mechanisms for specifying additional logical properties. The stack specification of Figure 1, for instance, might become the subject of the resource requirement: "This stack must be implemented by the general-purpose database system we already own". In this case no further design decisions need be made, and the requirement can be attached permanently to this interface specification.

The only properties actually specified in Figure 1 are those of a collection of elements with input and output operations, so the specification could also refer to a queue. Additional properties that guarantee stack-like behavior are introduced in Figure 2, which is an "elaboration" of Figure 1, meaning that primitives of Figure 1 have now been defined in terms of other primitives themselves. Any elaboration step makes some decisions that were previously deferred.

In Figure 2, the stack now has a maximum size of 10 elements, it is represented by an array and an index (used as the size and top pointer), and operations on it are defined in terms of functions such as

Figure 2. Elaboration of stack primitives.
The notation "j1< x ELEMENT >10" is equivalent to $\prod_{j=1}^{10}$ ELEMENT.
The function "equal" is an intrinsic predicate testing any two elements of the same set for equality.

```
STACK =
     ARRAY x INDEX

ARRAY =
     j1< x ELEMENT >10

INDEX =
     { 0,1,2,3,4,5,6,7,8,9,10 }

full:
     STACK
---> { true, false }

full((a,i)) =
     equal(i,10)

put-on:
     STACK x ELEMENT
---> STACK

put-on((a,i),e) =
     (indexed-insertion(a,increment(i),e),
      increment(i)
     )

indexed-insertion:
     ARRAY x INDEX x ELEMENT
---> ARRAY

     This function updates an array using an index to specify an element, and
     replacing the old element by the given one.

increment:
     INDEX
---> INDEX

     This function adds 1 to its argument.

empty:
     STACK
---> { true, false }

empty((a,i)) =
     equal(i,0)

take-off:
     STACK
---> STACK x ELEMENT

take-off((a,i)) =
     ((indexed-deletion(a,i),decrement(i)),
      indexed-retrieval(a,i)
     )
```

```
indexed-deletion:
     ARRAY x INDEX
---> ARRAY
```

This function updates an array using an index to specify an element, and deleting that element.

```
decrement:
     INDEX
---> INDEX
```

This function subtracts 1 from its argument.

```
indexed-retrieval:
     ARRAY x INDEX
---> ELEMENT
```

This function uses an index to specify an array element, and returns that element as its value.

"indexed-retrieval". The name, usage, and associated comment of "indexed-retrieval" all imply that indexing is used as the basic access mechanism for information in the stack. Thus this elaboration rules out the possibility of implementing the stack as a linked list with variable-size nodes scattered around a large workspace; it is effectively limited to being a list of fixed-size nodes in contiguous storage. But other decisions, such as what the stack elements will look like, are still deferred.

The array structure could have been motivated by a performance requirement such as "Any stack operation must be completed in less than 10 microseconds". The indexing mechanism which has now been specified guarantees that each access operation can be done in a fixed amount of time. The requirement can now be expressed precisely in the form of time limits on the executions of primitive access-operation components.

While the specification language records design decisions that have been made, it must avoid constraining decisions that have not. Implementation decisions--as opposed to design decisions--are avoided because this notation allows the designer to specify what a function does without specifying the data or control structures that will be used to do it. The definition $f(x,y) = g(h(x),j(y))$, for instance, could be implemented with parallel or serial evaluations of h and j; similarly, h(x) could be made available to g through a variable, a temporary buffer, or a network transmission. (Design decisions about these evaluation structures can also be specified functionally at the appropriate level of abstraction; there is a brief example of this at the end of Section 4).

In summary, functional specifications are distinguished from specifications of data abstractions by the notions that the performance and resource properties of an abstract concept can be as important as its logical properties in a large-scale development effort, and that these non-logical properties can only be associated meaningfully with a structural specification of the logical ones. At some point in the development of the system--presumably beyond the phases where asynchronous interactions are designed (i.e. early design), and certainly beyond the phases where design changes are observable to the user (i.e. requirements analysis)--the importance of storage and control structures becomes predominant, and data abstractions become important. Until then, whatever the designer decides is a "what" rather than a "how".

3. Asynchronously Interacting Processes

A digital device goes through a sequence of well-defined states. A process is a formal model of a digital device; it is here defined to have a state space (the set of all possible states of the device) and a successor function (or relation) on that state space that maps each state to its successor state(s).

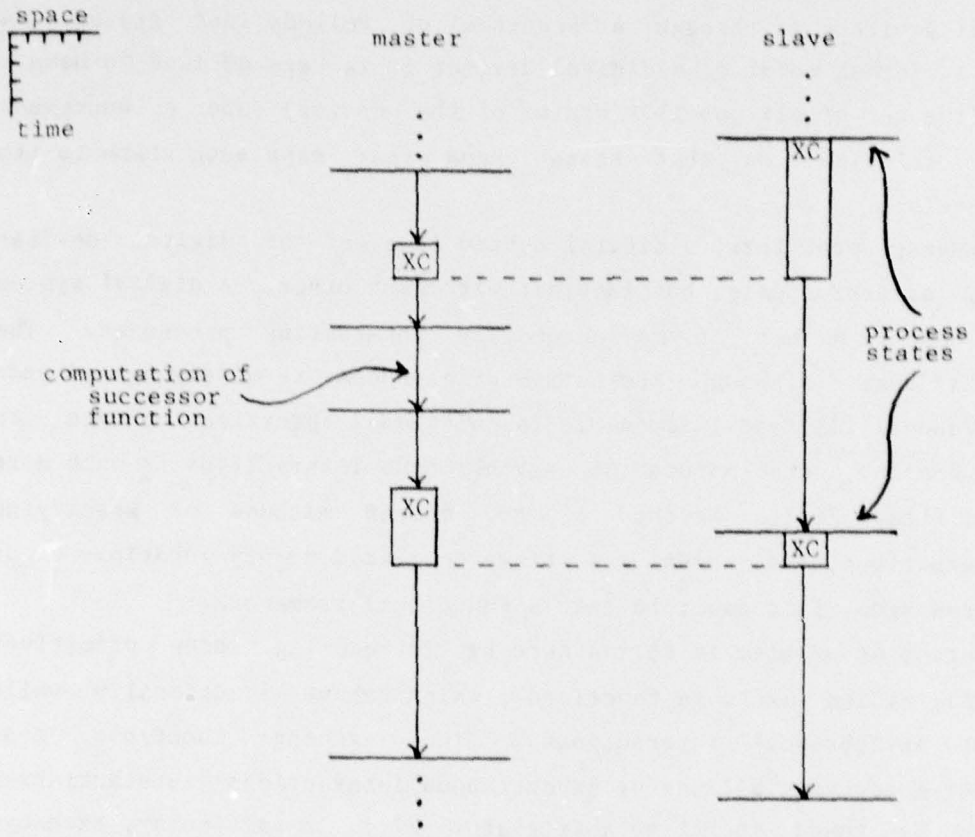
In its most general form, a digital system is a set of digital devices which operate asynchronously, but interact with each other. A digital system is specified here as a set of asynchronously interacting processes. The difficulty is that, although the nature of a process is widely agreed upon ([Horning & Randell 73]) and is amenable to functional specification (via its successor function), the nature of asynchronous interactions is much more controversial ([Zave 76]). Neither of the common methods of specifying process interactions, as reads and writes to shared memory locations or as transmitted messages, fits smoothly into a functional framework.

The interaction problem is solved here by introducing three primitives (XC, XA, XS), called "exchange functions", which behave "functionally" while carrying out asynchronous interactions. Since exchange functions seem sufficient for specifying all useful asynchronous interactions (substantiating examples can be found in [Zave & Fitzwater 77]; in particular, exchange functions are used to specify a standard message-passing mechanism), they make it possible to specify functionally digital systems of all varieties.

Consider two processes which interact in "master-slave" mode, in the sense that the only duty of the second process is to carry out orders from the first process. Their communication can be specified using the XC (eXchange to Communicate) function, as follows (see Figure 3).

At the beginning of each process step (i.e. computation of its successor function), the slave process initiates evaluation of XC('ready'). If the master does not yet have a command to give, evaluation of XC('ready') pauses and the slave waits. When the master does have a command to issue, it initiates evaluation of XC(<command>). At this point the evaluations exchange arguments and both terminate, so that the XC in the slave returns the <command> as its value, and the XC in the master returns 'ready'. This message tells the master that the slave has encountered no anomalous

Figure 3. "Master-slave" communication using XC.



conditions, and can be expected to carry out its orders without further intervention.

If the master initiates another XC(<command>) before the slave is finished, it will have to wait until the slave finishes its process step, begins another, and issues a matching XC.

Every instance of an exchange function belongs to a class (denoted by a subscript), which acts as a logical channel name. It can exchange only with other members of its class. Thus the master's exclusive use of the slave depends on the fact that only it evaluates exchange functions in the same class as the slave (its "ownership" of the slave is a matter of interpretation, since their communication is actually symmetric).

Now consider a process which models a CPU. The state space of the process is the set of all possible states of the processor, and each process

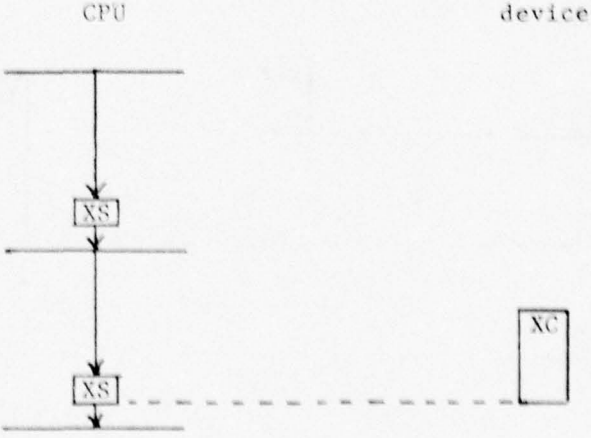
step represents the execution of one instruction. An interrupt mechanism can be specified for this processor using XC and the XS (eXchange Synchronous) function (see Figure 4).

The device causing the interrupt initiates evaluation of XCi(<interrupt-condition>). Near the end of each process step (instruction execution) the microcomputer process evaluates an XSi, using any argument which is distinguishable from an <interrupt-condition>. XSi behaves like XCi, except that it will not wait to be matched. If there is an XCi (i.e. an interrupt) pending, the two functions will exchange and return. If there is not, the XSi returns directly with its own argument as its value. The microcomputer process can determine the outcome from the value returned, and take appropriate action if there was an interrupt.

Finally, consider a set of processes which have need of a real-time clock. The clock itself is a process which "ticks" every time it takes a step. On each step it evaluates an XSt(<current-time>), thereby offering the current clock value to any process that wants it.

A process could read the clock by evaluating an XCt (argument irrelevant), but this leads to problems. If two processes ask for the clock value before the clock ticks again, their pending XCt's will exchange with each other! To prevent this we introduce a third function, XA (eXchange

Figure 4. An interrupt mechanism using XS.



Asynchronous). XA is the same as XC except that XA's cannot exchange with each other. If the two reading processes use XAt's, they will interact with the clock process as shown in Figure 5. Since, in this example, two processes can never read the same clock value, real times can be used for conflict resolution.

Figure 6 summarizes the possible interactions between exchange functions in the same class.

The real-time clock example shows that general implementation of these primitives requires conflict resolution in a distributed network, i.e. a means by which individual function evaluations can be matched into interacting pairs. Our only logical requirement for matching is that no pending exchange be locked out. In the real-time clock example, for instance, this "fair scheduling" rule could be implemented by sending notifications of all XA initiations to a queue at the clock's network node and matching the XA's to XS's in First-Come-First-Served order. This matching might not be FCFS in real time because of different transmission delays experienced by the XA initiation messages, but it would prevent lockout. It is also possible to do

Figure 5. Reading a real-time clock with XA.

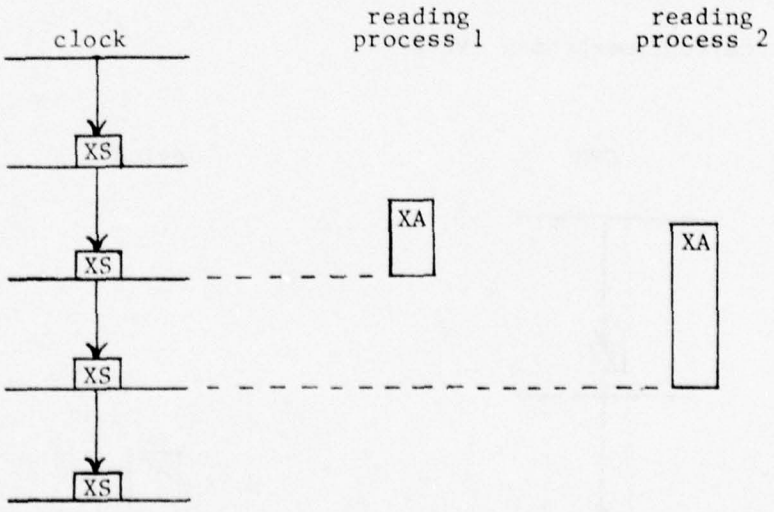
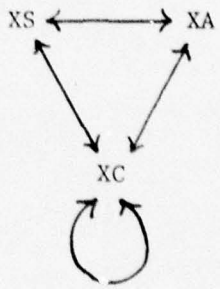


Figure 6. Possible interactions between exchange functions in the same class.



the matching FCFS in real time, to a known margin of error, as shown in [Lamport 78].

Exchange functions are exactly like other computable functions in that their evaluation is initiated with an argument, and terminates with the returning of a value. Being partial functions, XC's and XA's can fail to terminate--if they are never matched. In complex systems, use of these functions will have to satisfy sufficient, syntactically verifiable, conditions for termination (see [Zave & Fitzwater 77] for some early results). But unlike the other functions in a functional specification, exchange functions have side effects, through which they interact.

These primitives have some similarities to the input/output commands proposed in [Hoare 78], but the input/output commands are defined in a procedural, rather than functional, context. This distinction accounts for many subtle differences in the primitives. For instance, the logical channel names for input/output commands are the same as the corresponding process names; the program counter governing execution of the sequential program within the receiving process determines where a received value is to be stored. Several different exchange functions can be evaluated in parallel during a process step, on the other hand, and different classes are required to distinguish them. Furthermore, input/output commands have no analogue to the XS, and therefore cannot specify real-time systems.

4. System Specification

We will now apply functional specification techniques to subsystem and system structures. In Figure 7 the stack specification has been embedded in a "database process" of its own. The process is specified by its state space, which is the set "STACK", and by its successor function "stack-successor". An initial state which is the empty stack "(initial-array,0)" has been provided.

Each step of the process is an evaluation of its successor function "stack-successor". It begins with evaluation of the function "receive-command", which returns a "push" or "pop" command as its value. Let us assume for the moment that there is only one process (not specified here) in the system which is the source of these commands, and that it uses the syntax "send-command(create-command)". "Receive-command" is defined as an exchange function of type XC and class C, and is called with the argument "null". "Send-command" would also be defined as an exchange function of type XC and class C, but with the command as its argument.

The XC provides symmetric mutual synchronization and two-way communication. In this case, "receive-command" uses a null argument because it has no information to transmit; since its range is "COMMAND", the system specification will not be consistent unless the domain of "send-command" (and the range of "create-command"!) is "COMMAND". The opposite situation occurs when the stack process finishes the operation and returns the result to the user process (i.e. the source of the command, as above). The result is the argument of "send-reply", defined as an exchange function of type XC and class R. The user process would get the result by evaluating a function "receive-reply", defined as an exchange function with type XC, class R, and argument "null".

The definition of "new-stack" in the stack process is:

```
new-stack(s,r) =
     proj-2-1(s,send-reply(r)),
```

where s is the stack after the stack operation, r is the result of the operation, and "proj-2-1" projects this pair onto its first component. The first component provides the new state of the stack process, while "send-reply" is evaluated only for its side-effect of providing a reply to the user process--the null value it returns is thrown away.

Figure 7. Specification of a process.

A process is specified by its successor function, which is a function on its state space. An initial state value is specified as part of the successor function's domain description, e.g.:

```
successor-function:
        STATE-SPACE .. initial-state-value ..
    ---> STATE-SPACE
```

An exchange function of type XC and class Z is a primitive function with the name "xc-z", etc. The function which projects an n-tuple onto its mth component is a primitive function with the name "proj-n-m".

```
STACK-TYPE DATABASE PROCESS

COMMAND =
     OPERATION x DATA

OPERATION =
     { push, pop } ,

DATA =
     ELEMENT U NULL

NULL =
     { null } ;

RESULT =
     ACKNOWLEDGMENT U ANSWER


stack-successor:
     STACK .. (initial-array,0) ..
---> STACK

stack-successor(s) =
     new-stack
          (obey-command
               (s,receive-command))


initial-array:
---> ARRAY

     This function provides an initial array structure.


receive-command:
---> COMMAND

receive-command =
     xc-c(null)


obey-command:
     STACK x COMMAND
---> STACK x RESULT

obey-command(s,(o,d)) =
     [equal(o,push):  push(s,d),
      equal(o,pop) :  pop(s)
     ]
```

```
new-stack:
     STACK x RESULT
---> STACK

new-stack(s,r) =
     proj-2-1(s,send-reply(r))

send-reply:
     RESULT
---> NULL

send-reply(r) =
     xc-r(r)
```

This database specification uses a small number of primitive specification concepts, yet incorporates automatically the encapsulation of a data abstraction and the protection of a monitor ([Hoare 74], [Hoare 78]): All operations on the database are strictly sequenced, and the database can be accessed in no other way than through the controlled channels. The current trend toward such encapsulated structures strongly suggests that they are the best form in which to specify systems. Structures such as variables shared by several processes (the form in which the processes of a multiprogrammed system actually communicate) should arise only in the implementation of the higher-level structures, just as goto's should arise only in the compilation of structured programs.

At the same time that it offers all this interface structure, functional specification allows the deferment of many significant internal decisions. Consider this example: An update to a database may fail because resources of some kind (most likely space) are exhausted during the operation. Undoing a half-done operation, which has put the database into an inconsistent state, is usually difficult. There are several ways to deal with this, such as (a) working on a second copy of the database and not throwing the original away until the operation has been completed successfully, or (b) keeping a log of primitive operations which can be reversed to reconstruct the original database if necessary. The functional specification, which merely requires that a well-defined state be succeeded by a well-defined state, subsumes either approach. No procedural specification of the database operations could leave this decision deferred, however, because procedures to copy a data structure are quite different from procedures to modify it in place. Another example of a decision deferred by a functional specification is whether or not to exploit potential parallelism within the process (such as evaluation of a function's arguments in parallel).

Figure 8 completes the specification of a (trivial) real-time system of which the stack process of Figure 7 is a part. There is a "Producer Process", a "Consumer Process", and a "Real-Time Clock Process". The producer and consumer communicate with each other through the stack, i.e. the stack functions as a bounded LIFO message buffer between them. Thus the producer does all the "push" operations (see the function "send") and the consumer does all the "pop" operations (see "receive").

The producer and consumer processes compete for cycles of the stack process, which can receive and obey only one command per process step. If, when the stack process is ready to receive a new command, both producer and consumer have commands ready (i.e. have initiated evaluations of exchange functions in the class C and with the commands as their respective arguments), then only one of the exchanges matches the XC in the stack process--the other must wait until the next step of the stack process. Thus "send" and "receive" use XA's for the same reason as the competing processes of Figure 5.

Note that the producer and consumer do their own exception handling, based on individual characteristics. The "monitor" function in the producer checks the acknowledgment to see if the "push" was refused because the stack was full. If it was, the element is saved for retransmission on the next step, and the "FLAG" state component is set to "wait", delaying production of new elements. The consumer checks the validity of the answers it receives, but needs no buffering.

The real-time clock is the clock process of Figure 5. The function "new-time" (like "new-stack") is defined as a pair, only the first component of which forms the next state of the clock process. The other component is an exchange function, evaluated for its side-effect of offering the time to whomever wants to read it. This is a real-time system because the (so far unelaborated) functions "produce" and "consume" make use of the current time in some way which will be reflected by the values they generate.

The system specified in Figure 8 is "closed", i.e. has no external interactions. Because no parts are missing, it can be simulated according to the description in Section 2. Even more important, this specification can be interpreted as a specification of the requirements for a system. If the producer, consumer, and clock processes constitute the environment, for instance, then this is a very precise requirement for a database.

In the case of a far more complex database, this level of requirement

Figure 8. Specification of a real-time system.
The system has four processes, including the stack process.

```
PRODUCER PROCESS

PRODUCER-STATE =
     PRODUCER-DATA x (FLAG x ELEMENT)

FLAG =
     { proceed, wait }

producer-successor:
     PRODUCER-STATE .. (initial-producer-data,(proceed,e0)) ..
---> PRODUCER-STATE

producer-successor(d,(f,e)) =
     [equal(f,proceed):
          new-producer-state(produce(d,xa-t(null))),
      equal(f,wait):
          (d,send-and-check(e))
     ]


produce:
     PRODUCER-DATA x TIME
---> PRODUCER-DATA x ELEMENT

     This function prepares a message for the Consumer Process, and the next
     state of the Producer. Both values are dependent on real time.


new-producer-state:
     PRODUCER-DATA x ELEMENT
---> PRODUCER-STATE

new-producer-state(d,e) =
     (d,send-and-check(e))


send-and-check:
     ELEMENT
---> FLAG x ELEMENT

send-and-check(e) =
     monitor(e,check(send(e)))


send:
     ELEMENT
---> NULL

send(e) =
     xa-c((push,e))


check:
     NULL
---> ACKNOWLEDGMENT

check(null) =
     xc-r(null)


monitor:
     ELEMENT x ACKNOWLEDGMENT
---> FLAG x ELEMENT
```

```
monitor(e,a) =
     [equal(a,accepted):  (proceed,e0),
      equal(a,refused) :  (wait,e)
     ]
```

CONSUMER PROCESS

```
consumer-successor:
     CONSUMER-STATE
---> CONSUMER-STATE

consumer-successor(c) =
     consume-if-available(c,receive)


receive:
---> ANSWER

receive =
     xc-r(xa-c(pop,null))


consume-if-available:
     CONSUMER-STATE x ANSWER
---> CONSUMER-STATE

consume-if-available(c,a) =
     [equ(a,stack-is-empty):  c,
      true                 :  consume(c,a,xa-t(null))
     ]

wait:
     CONSUMER-STATE
---> CONSUMER-STATE
```

This function delays the Consumer so the Producer Process can catch up.

```
consume:
     CONSUMER-STATE x ANSWER x TIME
---> CONSUMER-STATE
```

This function consumes a message from the Producer. It is dependent on real time.

REAL-TIME CLOCK PROCESS

```
clock-successor:
     TIME
---> TIME

clock-successor(t) =
     new-time(tick(t))


tick:
     TIME
---> TIME
```

This function updates the real time by one "tick".

```
new-time:
     TIME
---> TIME

new-time(t) =
     proj-2-1(t,xs-t(t))
```

specification could establish the basic syntax of the query language, along with a bound on the number of simultaneous users, etc. Further elaborations of both the user processes and the database process could define completely the query/response language, and could establish as much detail about the structure of the database and the semantics of internal access functions as was necessary to guarantee the customer's satisfaction. This level is reached when the customer and designers agree that the meanings of all remaining primitives are well understood (or, better yet, are specified in terms of data abstractions). The sequence of elaborations used to reach this point provides a top-down, hierarchical structure for the requirements themselves.

Alternatively, the functions "produce" and "consume" of Figure 8 could be elaborated to show that they read sensors and emit control signals, respectively, for an industrial plant. This elaboration would produce at least one "Plant Process"; the plant and clock processes are then the environment. Real time could be used by "produce" to put time stamps on messages, and by "consume" to check how long they had been waiting.

These examples show how the benefits of formal specification, simulation, and other design tools can be extended to requirements analysis problems. They also show that functional specification techniques blur the line between requirements and early design phases--in many cases the same elaboration will refine both the requirements and the system design. In [Fitzwater & Zave 77] a patient-monitoring system is specified at several phases of development, including early requirements and resource-sharing models.

Thus the primitive notions of functional specification of asynchronous processes seem to be applicable to a wide variety of system types and development phases, yet retain the ability to specify powerful concepts (e.g. data abstraction/monitors such as the stack process) in a simple, natural way. This supports the idea that they are "basic building blocks" of system design, and could be used to specify, understand, and generate important system structures predictably. The ultimate goal of such knowledge would be to produce rules by which systems could be designed in incremental elaborations.

Note that this notion of functional elaboration is considerably more general than top-down design rules, such as stepwise refinement, defined on the program (procedural) domain alone. Within the scope of that domain the ideas may be identical, but methodologies based on procedures cannot extend across the abysses between different "viewpoints" (as in SADT, [Ross 77],

[Ross & Schoman 77]).

The typical digital system, for instance, can be seen from the viewpoints of (a) requirements, which can be expressed in the form of stimulus-to-response flows (as in the SREM project and its RSL requirements specification language, [Bell et al. 77], [Alford 77], [Davis & Vick 77]) to show the properties most visible to the user, (b) software, in which the importance of data and control structures makes procedures a good vehicle of communication, (c) resources, in which the properties of interest are allocation mechanisms and policies, performance, etc., and the application system is seen as a consumer, or (d) hardware, in which the system is a set of asynchronously interacting hardware components, and most of the properties we associate with the requirements have been encoded in values stored in the memory locations of program and data. Each of these viewpoints is distinguished by the set of properties that is represented explicitly. Within each viewpoint designs can be decomposed hierarchically, but expressions of different viewpoints tend to be incommensurate. This is one of the major difficulties of complex system design.

Fortunately, functional notation may be able to bridge these gaps. One example is that it can describe a process/interaction structure while still offering explicit representation of required stimulus-response paths, simply because the specification can be simulated to produce the required stimulus-response behavior.

For another example of gap-bridging, the storage properties characteristic of procedural languages can be introduced by functional elaboration as illustrated in Figure 9, which elaborates the "indexed-retrieval" function from the stack specification. The unique name of a data structure is transformed to a virtual address by "get-address". Via exchanges, "get-address" calls on a "Dynamic Allocation Process" to get its value. The dynamic allocator keeps the dynamic address mappings, and also performs such chores as compaction and garbage collection. If the "get-address" call were for the head of a list, for instance, the dynamic allocator might not have evaluated its address-returning "xc-addr" function until it had completed a garbage collection (by calls on the physical memory process) and revised its internal address mapping.

The function "read-element" in Figure 9 shows that an "element" is stored in two adjacent locations in virtual space. Properties concerning the

Figure 9. Elaboration of Figure 2 to provide explicit representation of storage properties and resource properties.

```
DATA-STRUCTURE =
     ARRAY U LIST

ELEMENT =
     WORD x WORD

indexed-retrieval:
     ARRAY x INDEX
---> ELEMENT

indexed-retrieval(a,i) =
     read-element
          (virtual-sum
               (get-address(name(a)),i))

name:
     DATA-STRUCTURE
---> NAME

     This function maps a particular instance of a data structure to its
     unique name.

get-address:
     NAME
---> VIRTUAL-ADDRESS

get-address(n) =
     xc-addr(xa-name(n))

virtual-sum:
     VIRTUAL-ADDRESS x INTEGER
---> VIRTUAL-ADDRESS

     This function computes a virtual address from a virtual address and a
     displacement.

read-element:
     VIRTUAL-ADDRESS
---> ELEMENT

read-element(v) =
     (read(v),read(virtual-sum(v),1))

read:
     VIRTUAL-ADDRESS
---> WORD

read(v) =
     xc-return(xc-access(v))
```

resource of memory are introduced with the definition of the "read" function. The exchanges evaluated in "read" communicate with a "Hardware Memory Process," which contains memory cells and the hardware virtual-physical

address map. This process translates the virtual address, fetches the word, and returns it. The Hardware Memory Process also communicates with a "Memory Allocation Process": specifically, the memory allocation process interrupts normal operations of the hardware memory process to change the virtual-physical map.

5. Related and Future Work

The ideas about requirements presented here have counterparts in several other approaches to requirements and design specification. We share with SADT the notion that requirements should be thought through carefully and checked with the customer, but SADT chooses the informal path, basing designer-customer communication on the fact that the specification technique imposes structure on requirements written in English. SADT also describes a system as a static collection of processors and data structures; the lack of explicit representation of sequence and behavior information limits its contribution to the solution of real-time design problems. We share with HOS ([Hamilton & Zeldin 76]) the formal, functional approach to specification, and with the SREM project the idea of an effective language admitting both logical and performance simulations of the specified system. However, since neither RSL nor HOS allows the explicit specification of system components such as processes and states, neither is helpful in designing these structures.

We share with the DREAM system ([Riddle et al. 78]) the specification of a high-level system design as a set of processes, with carefully structured asynchronous interactions. The most immediately apparent difference is that DREAM Design Notation assumes the use of mechanisms such as monitors, indicating less interest in basic building blocks than is shown here. This is because the DREAM system is somewhat specialized toward the software viewpoint, and one of the most important functions of DREAM Design Notation is to serve as a basis for an eclectic set of modeling (for feedback to the designer) and other practical design tools.

Our immediate goal, on the other hand, is to explore the properties and possibilities of our simple primitives. In the immediate future this will include (a) the study of requirements specifications, with particular attention to the formulation of performance and resource requirements, (b) the study of well-formed process structures, e.g. deadlock-free exchange patterns, and (c) the study of elaboration as a means by which layers of design (e.g. resource allocation mechanisms and policies) with predictable properties can be introduced incrementally.

The syntactic constructions introduced in this paper do not yet constitute a complete specification language. Macros and parameters should be

added for convenience. So should a mechanism for controlling the scope of local elaborations on different parts of a large specification. The resulting language could become the basis for a design database system with facilities for text editing, information retrieval, and collection of definitional blocks into complete specifications. This database could then support automated design tools.

Even in its present form, the specification language has already proven useful. It has been used to create and implement an innovative and promising design for a complex numerical system. The design is described in [Zave & Rheinboldt 79], and the formal specification is published in [Zave 78]. The numerical system has a high degree of parallelism and data segmentation, both of which are necessary to make an expensive but valuable numerical technique available for larger applications than are now possible. The form in which the parallelism appears was inspired by the asynchronous process structures described in Sections 3 and 4. The formal specification, in which all numerical processing is relegated to primitive functions, provides documentation and a clean interface between mathematical and systems programmers, none of whom need concern themselves with details of the other group's work.

6. Acknowledgments

Many of the ideas in this paper were motivated by conversations with D. R. Fitzwater. Their presentation has been much improved by suggestions from Richard G. Hamlet, Steven L. Small, and Marvin V. Zelkowitz.

7. References

[Alford 77]
Alford, Mack W. "A Requirements Engineering Methodology for Real-Time Processing Requirements." Trans. on Software Engineering SE-3, January 1977, pp. 60-69.

[Bell et al. 77]
Bell, Thomas E., Bixler, David C., and Dyer, Margaret. "An Extendable Approach to Computer-Aided Software Requirements Engineering." Trans. on Software Engineering SE-3, January 1977, pp. 49-60.

[Davis & Vick 77]
Davis, Carl G., and Vick, Charles R. "The Software Development System." Trans. on Software Engineering SE-3, January 1977, pp. 69-84.

[Fitzwater & Zave 77]
Fitzwater, D. R., and Zave, Pamela. "The Use of Formal Asynchronous Process Specifications in a System Development Process." Proceedings of the Sixth Texas Conference on Computing Systems, 1977, pp. 2B-21 - 2B-30.

[Goodenough 75]
Goodenough, John B. "Exception Handling: Issues and a Proposed Notation." Comm. of the ACM 18, December 1975, pp. 683-696.

[Hamilton & Zeldin 76]
Hamilton, Margaret, and Zeldin, Saydean. "Higher-Order Software--A Methodology for Defining Software." Trans. on Software Engineering SE-2, March 1976, pp. 9-32.

[Hoare 74]
Hoare, C. A. R. "Monitors: An Operating System Structuring Concept." Comm. of the ACM 17, October 1974, pp. 549-557.

[Hoare 78]
Hoare, C. A. R. "Communicating Sequential Processes." Comm. of the ACM 21, August 1978, pp. 666-677.

[Horning & Randell 73]
Horning, J. J., and Randell, B. "Process Structuring." Computing Surveys 5, March 1973, pp. 5-30.

[Lamport 78]
Lamport, Leslie. "Time, Clocks, and the Ordering of Events in a Distributed System." Comm. of the ACM 21, July 1978, pp. 558-565.

[Liskov & Zilles 75]
Liskov, Barbara H., and Zilles, Stephen. "Specification Techniques for Data Abstractions." Trans. on Software Engineering SE-1, March 1975, pp. 7-19.

[Riddle et al. 78]
Riddle, William E., et. al. "Behavior Modeling During Software Design." Trans. on Software Engineering SE-4, July 1978, pp. 283-292.

[Ross 77]
Ross, Douglas T. "Structured Analysis (SA): A Language for Communicating Ideas." Trans. on Software Engineering SE-3, January 1977, pp. 16-34.

[Ross & Schoman 77]
Ross, Douglas T., and Schoman, Kenneth E. "Structured Analysis for Requirements Definition." Trans. on Software Engineering SE-3, January 1977, pp. 6-15.

[Zave 76]
Zave, Pamela. "On the Formal Definition of Processes." Proceedings of the International Conference on Parallel Processing, 1976, pp. 35-42.

[Zave 78]
Zave, Pamela. "The Formal Specification of an Adaptive, Parallel Finite Element System." Univ. of Md. Comp. Sci. Dept. TR-715, 1978.

[Zave & Fitzwater 77]
Zave, Pamela, and Fitzwater, D. R. "Specification of Asynchronous Interactions Using Primitive Functions." Univ. of Md. Comp. Sci. Dept. TR-598, 1977.

[Zave & Rheinboldt 79]
Zave, Pamela, and Rheinboldt, Werner C. "Design of an Adaptive, Parallel Finite-Element System." Trans. on Mathematical Software 5, March 1979, pp. 1-17.